POLITECNICO DI TORINO
Repository ISTITUZIONALE

IEEE Standard 1500 Compliance Verification for Embedded Cores

Original
IEEE Standard 1500 Compliance Verification for Embedded Cores / Benso A.; Di Carlo S.; Prinetto P.; Zorian Y.. - In: IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION (VLSI) SYSTEMS. - ISSN 1063-8210. - STAMPA. - 16:4(2008), pp. 397-407.

Availability:
This version is available at: 11583/1785476 since:

Publisher:
IEEE

Published
DOI:10.1109/TVLSI.2008.917412

Terms of use:
openAccess

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

05 August 2020

IEEE Standard 1500 Compliance Verification for Embedded Cores

Alfredo Benso, *Senior Member, IEEE*, Stefano Di Carlo, *Member, IEEE*, Paolo Prinetto, and Yervant Zorian, *Fellow, IEEE*

***Abstract*—Core-based design and reuse are the two key elements for an efficient system-on-chip (SoC) development. Unfortunately, they also introduce new challenges in SoC testing, such as core test reuse and the need of a common test infrastructure working with cores originating from different vendors. The IEEE 1500 Standard for Embedded Core Testing addresses these issues by proposing a flexible hardware test wrapper architecture for embedded cores, together with a core test language (CTL) used to describe the implemented wrapper functionalities. Several intellectual property providers have already announced IEEE Standard 1500 compliance in both existing and future design blocks. In this paper, we address the problem of guaranteeing the compliance of a wrapper architecture and its CTL description to the IEEE Standard 1500. This step is mandatory to fully trust the wrapper functionalities in applying the test sequences to the core. We present a systematic methodology to build a verification framework for IEEE Standard 1500 compliant cores, allowing core providers and/or integrators to verify the compliance of their products (sold or purchased) to the standard.**

***Index Terms*—Electronic design automation (EDA) tools, IEEE 1500 Standard, unified modeling language (UML), verification of embedded cores.**

I. Introduction

THE INCREASED density and performance of advanced silicon technologies made system-on-a-chip (SoC) application-specific integrated circuits (ASICs) possible. SoCs bring together a set of functions and technology features on a single die of enormous complexity. Each component is available as a predesigned functional block that comes as an intellectual property (IP) embedded core, reusable in different designs. These so-called embedded cores make it easier to import technologies to a new system and differentiate the corresponding product by leveraging IP advantages. Most importantly, the use of embedded cores shortens the time-to-market for new systems thanks to a heavy design reuse [1].

What makes designing systems with IP cores an attractive methodology (e.g., design reuse, heterogeneity, reconfigurability, and customizability) also makes testing and debugging of these systems a very complex challenge [2].

Manuscript received March 2, 2007.

A. Benso, S. Di Carlo, and P. Prinetto are with the Department of Information and Automation Technologies, Politecnico di Torino, 10129 Torino, Italy (e-mail: alfredo.benso@polito.it; stefano.dicarlo@polito.it; paolo.prinetto@polito.it).

Y. Zorian is with the Virage Logic, Fremont, CA 94538 USA (e-mail: zorian@computer.org).

Digital Object Identifier 10.1109/TVLSI.2008.917412

There exist strong functional similarities between the traditional system-on-board and the SoC design. However, their manufacturing test process is quite different.

In a system-on-board, the IC provider is responsible for the design, manufacturing, and testing of the ICs components of the system. In this context, the system integrator is, with respect to testing for manufacturing defects, only responsible for the interconnections between the ICs. The boundary scan test, also known as JTAG or the IEEE Standard 1149.1 [3], is a well-known technique to address this board-level interconnect test problem. In the SoC design flow, the core provider delivers a description of the core design to the system integrator at different possible levels: *soft* (register-transfer level), *firm* (netlist), or *hard* (technology-dependent layout). Being the core delivered only as a model, a manufacturing test is at this stage impossible. Therefore, the test responsibility of the system integrator now not only concerns the interconnect logic and wiring between the cores, but also the IP cores themselves.

For an SoC design, the test of embedded cores constitutes a large part of the overall IC test, and hence substantially impacts the total IC quality level as well as its test development effort and associated costs. The adoption and design of adequate test and diagnosis strategies is therefore a major challenge in the production of SoCs.

The IEEE Standard Testability Method for Embedded Core-Based Integrated Circuits (IEEE Standard 1500 [4]) addresses the specific challenges that come with testing deeply embedded reusable cores supplied by diverse providers, who often use different hardware description levels and mixed technologies [2], [5], [6]. The IEEE Standard 1500 defines a scalable standard architecture to facilitate and support the testing of embedded cores and the associated interconnect circuitry in a SoC.

The standard does not cover the core's internal test methods or chip-level test access configuration. The standardization effort focuses on non-merged cores (cores that are tested as standalone units) and provides the following two main supports:

- standardized *core test language* (CTL), capable of expressing all test-related information that need to be transferred from the core provider to the core user;
- standardized (but configurable and scalable) *core test wrapper*, allowing easy test access to the core in an SoC design.

Several publications presented solutions to build SoCs with IEEE Standard 1500 testability features [7], [8]; nevertheless, by analyzing the IEEE Standard 1500, it is clear that implementing a fully compliant core is not trivial. The IEEE Standard 1500 is in fact articulated in a large set of architectural rules. Some of them are very specific on particular design aspects, whereas

1063-8210/$25.00 © 2008 IEEE

others do not introduce particular restrictions but define general characteristics of the design. It is very easy therefore to forget one of the specific rules or misunderstand a general rule, implementing it with a custom architecture that does not respect the standard as a whole.

Verifying the actual compliance of a wrapped core to the IEEE Standard 1500 is therefore mandatory. Nevertheless, without a formalized approach, this verification task can be more expensive than the design of the core itself.

Looking at the available literature, only a few authors tried to address the problem of verifying the compliance of an IP core to the IEEE Standard 1500. In [9], Diamantidis *et al.* present an approach based on a dynamic, constrained-random coverage-driven verification methodology to verify the functionality of the complete test infrastructure within a given SoC. The main drawback of this contribution is that the authors verify the SoC and wrapper functionalities without systematically addressing every single aspect (rule) of the standard.

This paper shows a systematic methodology to build a verification framework for IEEE Standard 1500 compliant cores. This methodology does not aim at providing a complete implementation of the verification framework but it focuses on defining an abstract model that enables core providers and/or integrators to build their custom verification frameworks to verify the compliancy of their products (sold or purchased) to the IEEE Standard 1500. The model guarantees to systematically address the different aspects of the standard. It is in fact an abstraction of the standard itself and could be reused to build compliance verification frameworks for different standards as long as they are structured similarly to the IEEE Standard 1500 (e.g., the JTAG standard).

This paper is organized as follows. Section II overviews the basic concepts of the IEEE Standard 1500, whereas Section III introduces the basic elements composing the proposed IEEE Standard 1500 Verification Framework. Sections IV and V detail the different types of verification needed to build the framework. Finally, Section VI presents a prototype of the IEEE Standard Verification Framework and Section VII summarizes the main contributions and outline future research activities.

II. An Overview of the IEEE Standard 1500

This section introduces the main features of the IEEE Standard 1500 that will be extensively used in this paper.

The IEEE Standard 1500 defines a scalable standard design-for-testability architecture to facilitate testing and diagnosis of embedded cores and associated circuitry in an SoC. The architecture is independent from the underlying core's functionality and technology.

The IEEE Standard 1500 architecture always includes the following.

- *Core Test Wrapper*: A wrapper placed around the boundaries of the core that allows accessing its testing functionalities using a standard interface and protocol (see Fig. 1). The wrapper is completely transparent when the core is not in test mode.
- *Information Model*: A formal description of the IEEE Standard 1500 functionalities (mandatory and optional) implemented by the core test wrapper. The Information Model is the bridge between core providers and core users and facilitates the automation of test data transfer and reuse between these two entities. The Information Model is described using the IEEE 1450.6 CTL [10]. The information model includes the following:
  - set of wrapper's signals;
  - timing of the wrapper signals (wrapper communication protocol);
  - information about the test patterns.

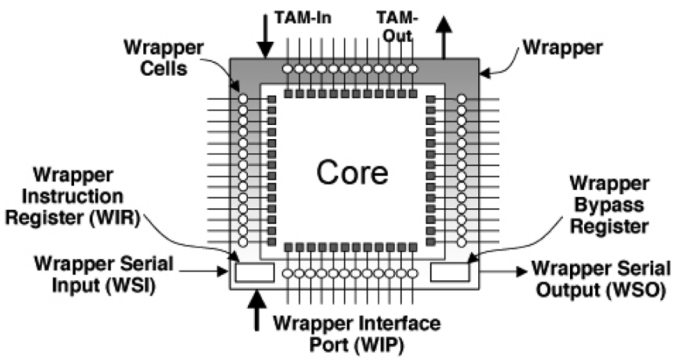

Fig. 1. IEEE 1500 core test wrapper architecture.

Fig. 1 shows the overall architecture of a general IEEE 1500 wrapper. It includes the following elements:

- *wrapper instruction register* (WIR) for controlling the wrapper operational mode;
- chain of wrapper cells called *wrapper boundary register* (WBR) to provide test functions at the core terminals;
- *wrapper bypass register* (WBY) for synchronously bypassing the wrapper;
- *wrapper interface port* (WIP) for serially controlling the wrapper using the wrapper serial input (WSI) and the wrapper serial output (WSO), and, optionally, a test access mechanism (TAM).

The IEEE Standard 1500 is an effort of reconciling and accommodating different test strategies and motives. The greatest effort has been put into supporting as many requirements as possible while still producing a cohesive and consistent standard.

In addition to the mandatory elements, the core designer is free to define a set of wrapper defined registers (WDR) and/or core defined registers (CDR). WDRs and CDRs are the mechanism used by the IEEE Standard 1500 to accommodate the different test strategies coming from different core providers.

The strong effort put into providing high flexibility can be finally translated into the definition of the following two levels of compliance to the standard.

- *IEEE Standard 1500 Compliant Core*: This notion refers to a core that incorporates an IEEE Standard 1500 wrapper function and comes with an IEEE Standard 1500 CTL program. The CTL program describes the core test knowledge, including how to operate the wrapper, at the wrapper's external terminals.
- *IEEE Standard 1500 Ready Core*: This notion refers to a core which does not have a complete IEEE Standard 1500 wrapper, but does have a IEEE Standard 1500 CTL description. The CTL description can be used to synthesize an IEEE Standard 1500 compliant wrapper to make the core

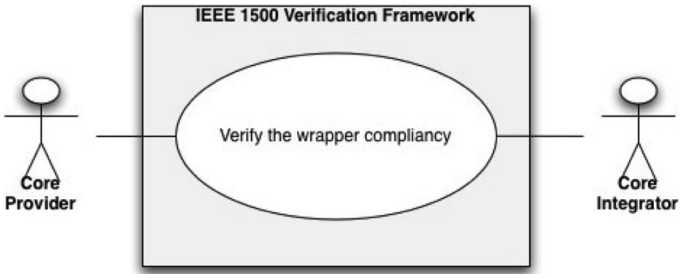

Fig. 2. IEEE 1500 Verification Framework UML use cases diagram.

fully compliant. The CTL program describes the core test knowledge at the bare core terminals.

III. Building the IEEE Standard 1500 Verification Framework

This section introduces the basic concepts needed to build a verification framework for the IEEE Standard 1500. Wherever possible, the framework will be described resorting to the unified modeling language (UML) [11]. UML is a semiformal specification language standardized by the object management group (OMG) [12]. The usage of UML allows building a model of the framework not biased towards a specific software implementation.

The first step to perform is the identification of the different scenario where the framework may be used. Fig. 2 shows the UML use cases diagram for the IEEE Standard 1500 Verification Framework.

The verification framework represents the system to model (identified by a rectangle in the diagram) and for this system we have a single use case (identified by the oval in the diagram) consisting in the verification of the compliance of a core test wrapper with the IEEE Standard 1500.

An UML use case defines a sequence of interactions between one or more actors, i.e., candidate users of the system, and the system itself. For the IEEE Standard 1500 Verification Framework, we envision the following two different actors.

- *Core Provider*: Core providers have a strong interest in providing IEEE Standard 1500 compliant designs to facilitate the integration of their cores into system-level test infrastructures. From the core provider perspective, selling a product as IEEE Standard 1500 compliant when the core is actually not fully compliant, is a high risk situation. Verifying the standard compliance after the core design has been signed-off can be really complex. In fact, if an error has been originated by an incorrect application of a rule, during the simulation and verification phase of the core the designer will very likely be unable to recognize the problem. Moreover, in case of errors appearing in corner cases of the core functionalities, the error can easily escape the debug phase but can be excited by the user application.
- *Core Integrator*: Core integrators need to be sure that IEEE Standard 1500 compliant cores actually comply with the required functionalities at both standalone and system-level. In case of errors in the wrapper design, the whole SoC compliancy or even functionality may be compromised. Moreover, since core integrators usually have a very poor knowledge of the core's internal structure, the verification of the IEEE Standard 1500 compliance may be even harder than for a core provider.

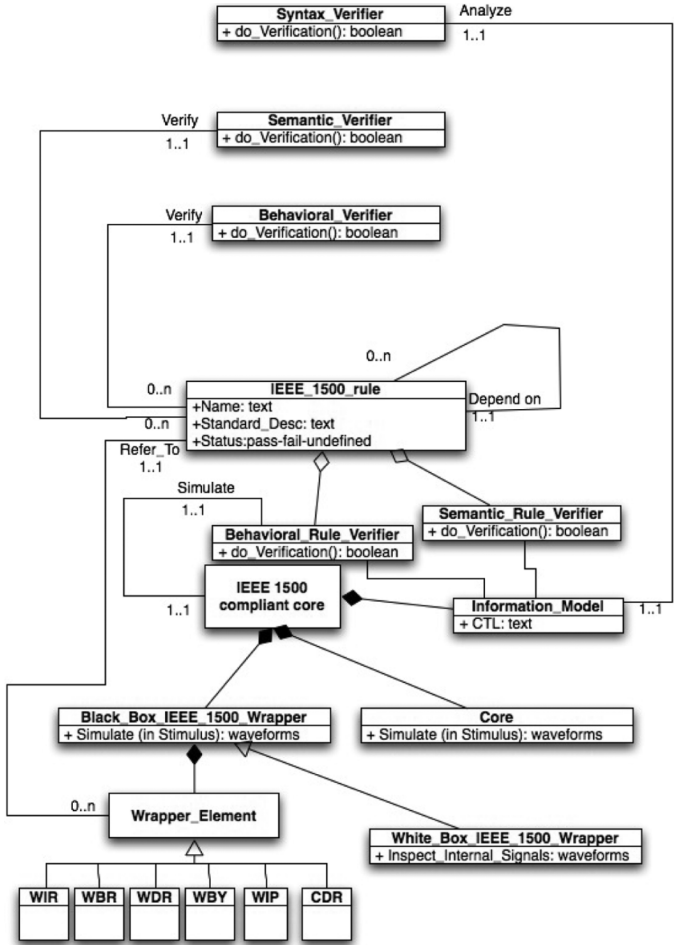

Fig. 3. IEEE 1500 Verification Framework UML class diagram.

Modelling the IEEE Standard Verification Framework means defining the actions performed by the *"Verify the wrapper compliance"* use case and defining the elements involved in those actions. The elements building the framework are modeled by an UML Class Diagram. A class diagram partitions the system into areas of responsibility (*classes*), and shows dependencies (*associations*) between them.

Fig. 3 depicts the class diagram for the full IEEE Standard 1500 Verification Framework. The different elements (classes and associations) of the diagram will be deeply explained in Sections IV–VI and will be used to formalize the operations performed by the framework.

Let us now consider the structure of the IEEE Standard 1500. As already introduced in Section II, the standard defines a core test wrapper and an information model defined as a set of CTL [10] statements.

The information model is a key element of the standard. To verify the compliancy of a core test wrapper with the IEEE Standard 1500, we have to first verify the correctness of the syntax of the CTL description provided with the wrapper. This simple constraint identifies the first element of the framework: a module in charge of performing a syntax analysis of the information

model. This module is modeled by a class (*Syntax_Verifier*) in the class diagram of Fig. 3.

Beside the language used to specify the information model, the IEEE Standard 1500 is then composed of a set of different rules that define how an IEEE Standard 1500 compliant core test wrapper has to be designed. Rules identify the following two different aspects of the standard.

- *Semantic Aspects*: They mainly concern the information model (CTL description) and can be verified without any interaction with the actual core/wrapper implementation (i.e., without any need of performing core/wrapper simulations).
- *Behavioral Aspects*: They target the communication protocols and the behavior of the wrapper. In general, the verification of these aspects requires a functional simulation of the wrapper.

The way semantic and behavioral aspects can be verified is quite different; in the framework, they require two separate modules in charge of performing the two types of verification. The two modules are modeled by the *Semantic_Verifier* class and the *Behavioral_Verifier* class of the diagram in Fig. 3.

At this point, we have three main modules in the IEEE Standard 1500 Verification Framework that correspond to the following three verification processes:

- *syntax verification*;
- *semantic verification*;
- *behavioral verification*.

Before adding more details to the model, it is important to define a possible verification plan in order to highlight dependencies between the results of the different verification steps. Moreover, being that the verification process is one of the main cost factors of a modern SoC, we have to define optimal verification plans targeting the reduction of the overall verification time.

Fig. 4 shows the verification plan for the syntax, semantic, and simulative verification in the proposed framework. It is modeled using an UML sequence diagram describing how groups of *objects* (instances of classes) collaborate to complete a given task. Typically, a sequence diagram captures the behavior of a single use case and in this particular case, it models the *"Verify the wrapper compliance"* use case of Fig. 2. The collaboration between the objects is represented by an exchange of *messages* between objects (calls to UML classes methods).

The verification process is managed by one of the actors of the model. At this abstraction level there is no difference between the two classes of users identified in the use case diagram of Fig. 2 (core integrator and core provider). As already introduced in this section, being that the information contained in the information model is an essential element for any other verification action, the first action to perform is the syntax verification.

The action of performing the verification is modeled as an exchange of a *do_Verification()* message between the Actor and the Syntax_Verifier class. The Syntax_Verifier performs the required checks and returns a Boolean value (*OK_Syn*) to the actor indicating whether the verification was successful or not. If the syntax verification fails, it is not possible to proceed with the next steps. The verification process ends and the core test wrapper under analysis is considered not IEEE Standard 1500 compliant.

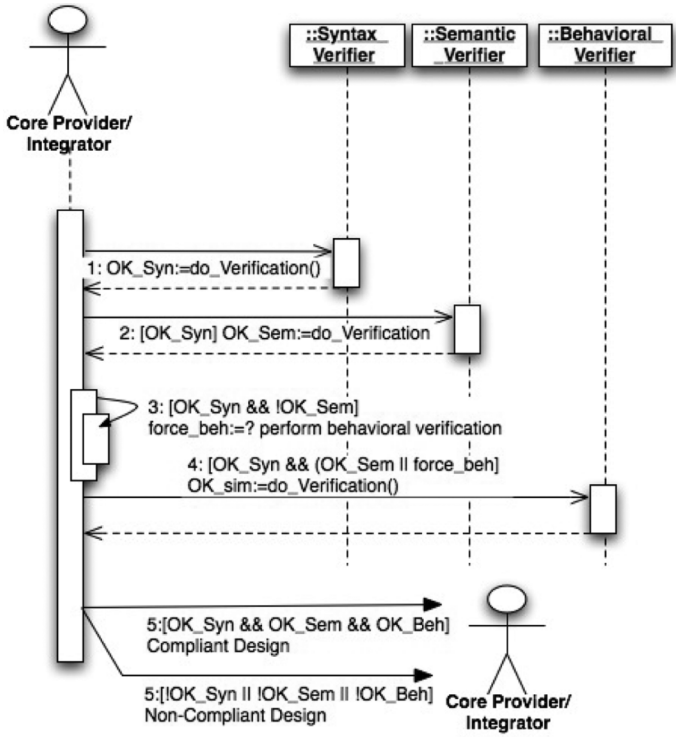

Fig. 4. IEEE 1500 UML Verification Framework basic verification plan.

If the syntax verification passes, it is then possible to move to the next verification phases. In order to reduce the verification effort in the case of non-compliant cores, the next scheduled verification is the semantic verification. This verification works on information contained in the information model (does not need a functional simulation of the wrapper) and can therefore be performed much faster than the behavioral verification. Moreover, the behavioral verification needs to use the information model, which than needs to be analyzed first.

Again the semantic verification starts with a *do_Verification()* message exchanged between the actor and the Semantic_Verifier class. The result of the message is a Boolean value (*OK_Sem*) stating whether the verification passed or failed.

In the case of a positive response from the semantic verification, it is possible to perform the last verification step (behavioral verification) modeled in Fig. 4 as a message exchange between the actor and the behavioral verifier. Moreover, in some situations it may be useful to perform the behavioral verification even in the case of a negative response from the semantic verification in order to better diagnose the cause of the non-compliancy. This possibility is modeled in Fig. 4 by the self-message *"?Perform behavioral verification"* sent by the actor to itself.

In Sections IV–VI, to complete the definition of the IEEE Standard 1500 Verification Framework, each verification step modeled in Fig. 4 will be analyzed in details.

IV. Syntax Verification

As introduced in Section III, the syntax verification is the first action performed by the IEEE Standard 1500 Verification Framework (see Fig. 4).

This analysis verifies that the information model (CTL) [10] provided together with an IEEE Standard 1500 compliant core is syntactically correct. The syntax verification does not deal

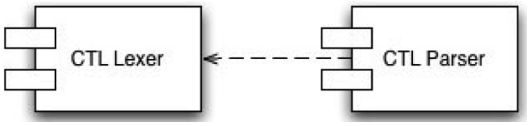

Fig. 5. IEEE 1500 syntax verification UML component diagram.

with the information contained in the information model, it just checks the syntax of the statements in the model.

The problem of verifying the syntax of a set of statements, according to a given language definition, is a well-known problem in the field of programming language compilers. It is usually solved using special programs called *lexers* and *parsers* [13].

In computer science and linguistics, parsing is the process of analyzing a sequence of tokens to determine its grammatical structure with respect to a given formal grammar. A parser is the component of a compiler that carries out this task. Parsing transforms input text into a data structure, usually a tree, which is suitable for later processing and which captures the implied hierarchy of the input. Lexical analysis creates tokens from a sequence of input characters. Tokens are processed by the parser to build a data structure such as parse tree or abstract syntax trees.

The syntax verification is modeled by the *Syntax_Verifier* class (see Fig. 3) in the IEEE Standard 1500 Verification Framework. The class implements the *do_Verification()* method actually performing the syntax analysis of the CTL provided with the information model. We can therefore say that: "the Syntax_Verifier class *analyzes* the syntax of the information model." This dependency is modeled in Fig. 3 by an association (*Analize*) between the Syntax_Verifier class and the Information_Model class that models the IEEE Standard 1500 information model.

The Syntax_Verifier class is finally composed of two main elements: 1) a *CTL lexer* and 2) a *CTL parser* that collaborate to analyze the syntax of the IEEE Standard 1500 information model. This property is modeled by the UML component diagram of Fig. 5 depicting the software components actually composing the Syntax_Verifier.

In order to implement the syntax verification it is enough to implement a parser and a lexer for the CTL language. Even if the CTL is an extension of the STIL language [14] and a standard itself [10], this implementation is not trivial since most of the lexical rules that define the CTL language are described using natural language and they have to be translated into a formal grammar allowing the implementation of the lexer and parser components.

V. IEEE Standard 1500 Rules Verification

The syntax verification, introduced in Section IV, is able to guarantee the syntactic correctness of the IEEE Standard 1500 information model only. In order to guarantee the compliancy of a core test wrapper with the IEEE Standard 1500 it is necessary to perform a deeper analysis. As introduced in Section III, this analysis has to be at the same time semantic, by using information obtained from the information model itself, and behavioral, by performing simulations of the wrapper using electronic design automation (EDA) tools. Both semantic and behavioral verification aim at proving that the features implemented in a core test wrapper are compliant with the definitions of the standard.

```
Rule 17.2.1.d:
"All digital terminals of IEEE 1500
wrapped or unwrapped cores with
corresponding wrapper cells shall
be identified in CTL using the
following statement :

Internal {
 signame {
  IsConnected {
   Wrapper IEEE1500 CellID cell_type;
  }
 }
}

Rule 11.3.1.d:
"While the WBY is selected and when
the signal connected to the WSCs
SelectWIR and ShiftWR terminals are
logic 0 and 1 respectively, a WBY
shift operation shall occur on the
next rising edge of WRCK".
```

Fig. 6. IEEE Standard 1500 rules examples.

The IEEE Standard 1500 basically consists in a collection of rules. The concept of *"IEEE 1500 rule"* is a key point to model the semantic and behavioral verification. Fig. 6 shows an example of two different rules.

From the previous example, it is clear that some rules target the semantic of the IEEE Standard 1500 information model (e.g., Rule 17.2.1.d of Fig. 6), whereas other rules focus more on the architectural or functional aspects of a component (e.g., Rule 11.3.1.d of Fig. 6). There are also rules that have both characteristics. We therefore identify the following three different rule categories:

- *semantic rules*;
- *behavioral rules*;
- *mixed rules*.

Each IEEE Standard 1500 rule is modeled in the IEEE Standard 1500 verification framework by the *IEEE_1500_rule* class (see Fig. 3). This class allows the formalization of a set of concepts expressed in natural language (english) by the standard. Each rule is characterized by the following attributes (see Fig. 3).

- *Name*: The rule's number as it appears in the IEEE Standard 1500 [4].
- *Standard_Desc*: The rule explanation as it appears in the IEEE Standard 1500 (in natural language).
- *Status*: Identifies (for a given core test wrapper) whether the rule has been verified with success, it failed, or still has to be verified.

Each rule targets different architectural and functional aspects of the core test wrapper. This dependency is modeled by the *Refer_To* association between the IEEE_1500_Rule class and

the Wrapper_Element class in Fig. 3. The Wrapper_Element class is an abstract class that identifies a general part of the wrapper. It is then specialized into the different actual components of the wrapper, e.g., WIR, WBR, etc. (see Section II).

Finally, each rule needs to have a verifier able to proof the correct implementation of the rule in the core test wrapper. From the classification of IEEE 1500 rules into semantic, behavioral, and mixed rules, we can identify the following two categories of rule verifiers.

- *Semantic Rule Verifiers*: In charge of verifying the semantic aspects of a rule.
- *Behavioral Rule Verifiers*: In charge of verifying the architectural and functional aspects of a rule.

The two types of rule verifiers are modeled in the framework with the *Semantic_Rule_Verifier* class and the *Behavioral_Rule_Verifier* class, respectively (see Fig. 3). The modeling and implementation of these two rule verifiers will be deeply analyzed in the following two subsections.

As already introduced in Section III, one of the requirements in the implementation of an IEEE Standard 1500 Verification Framework is the optimization of the verification process. In particular, it is necessary to understand whether, upon a rule failure, it is necessary to abort the whole verification process or not. To address this issue, we introduced a very detailed rule hierarchy. This hierarchy is represented in Fig. 3 as the *Depend_On* association. *Depend_On* is a one to many association that creates a relation between a rule and a set of other rules: if rule A depends on rule B, it means that in order to verify the compliancy to rule A it is necessary to first verify the compliancy to rule B. The identification of the optimal hierarchy is a key point in the definition of the verification plan and in the reduction of the verification costs. For the implementation of our prototype (see Section VI), to create the rule hierarchy, we took into account the following characteristics.

- *Verification Effort*: The time required to verify the rule. Faster rules (e.g., semantic rules) are, if possible, placed higher in the hierarchy.
- *Component Complexity*: The complexity of the components targeted by the rule. Rules that, in case of failure, require to fix the core itself are placed higher in the hierarchy and are usually critical for the continuation of the verification process.
- *Component Isolation*: The impact that the components targeted by the rule have in the wrapper. If the component is functionally isolated from the rest of the wrapper, then it is possible to continue at least part of the remaining verification process even if the rule fails.

An example of part of the rule hierarchy is presented in Fig. 7.

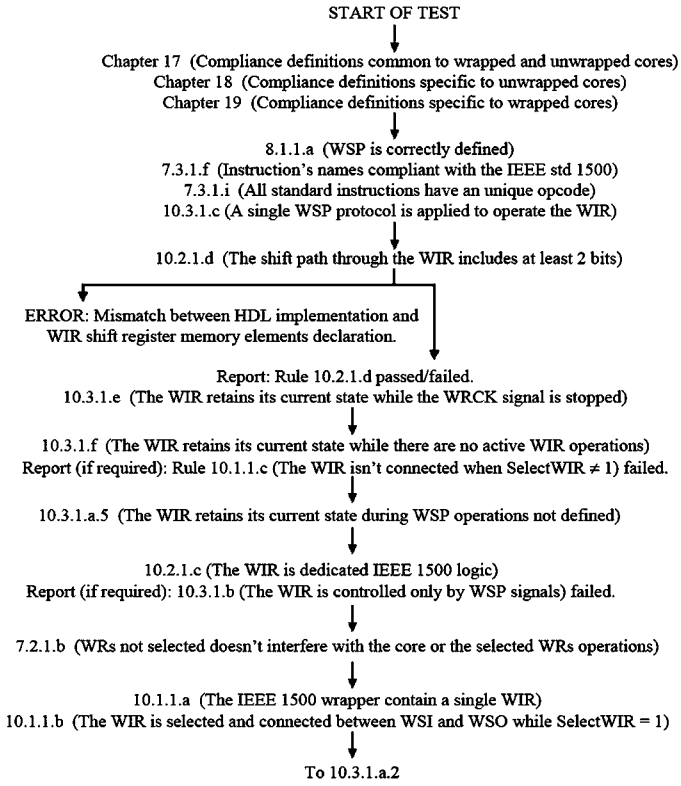

Fig. 7. Example of IEEE Standard 1500 rule hierarchy.

A. Semantic Rule Verifier

Semantic rules and part of the mixed rules can be verified by simply analyzing the content of the IEEE Standard 1500 information model provided with the core test wrapper. This is a fast and powerful way to verify at least part of the IEEE Standard 1500 compliancy since it does not require any simulation of the wrapper/core itself. Although simple and fast, this analysis is by no means enough to guarantee IEEE Standard 1500 compliancy. First of all, pure semantic rules are only a relatively small subset of the whole rules set. Moreover, the semantic analysis is performed on data contained in an information model supplied by the core provider; there is no guarantee that the CTL description perfectly matches the actual hardware implementation.

Looking at the content of the IEEE Standard 1500 information model the following types of CTL statements are responsible for most of the required semantic information:

- ScanStructures;
- MacroDefs;
- environments.

The main issue with the information model is that it may include many user-defined structures; also, many mandatory or optional hardware structures may be mapped on hardware components already present in the core and therefore using different naming conventions. Nevertheless, in order to be useful to the semantic verifier, the semantic information needs to be organized in a more formal and less core-dependent way. To overcome this problem, we propose to translate the CTL description related to a particular core implementation into a more general metadata model that can be more easily analyzed and proofed.

This operation can be efficiently performed during the parsing operation executed at the beginning of the syntax verification step (see Section IV). In particular, the semantic information of the information model has to be translated into the metadata structure and then renamed according to an internal and core-independent naming convention. In this way, all user-defined structures (and therefore signal/register names) can be mapped to their corresponding general templates into

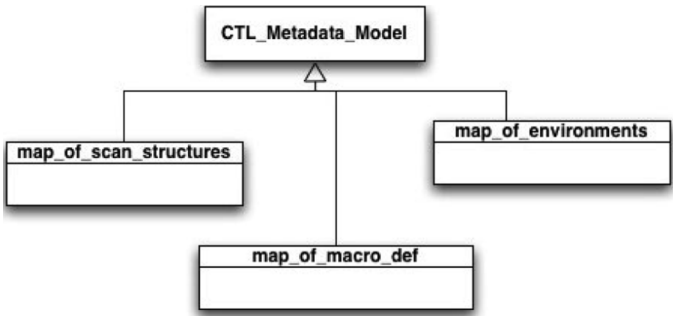

Fig. 8. IEEE 1500 Metadata UML class diagram.

the metadata model and all the semantic checks can now be performed on the metadata model, where the naming convention is independent from the corresponding core.

A very high level UML class diagram modeling an example of the metadata model is provided in Fig. 8.

Thanks to its internal naming convention and corresponding signal mapping, it becomes much easier to run checks on the content of the metadata model. For example, a simple rule that says

$$The\ WS\_Bypass\ instruction\ is\ mandatory$$

could be easily verified performing a control like

```
isset (map_of_ctl_filename (WS_Bypass))
```

This operation will automatically check if a mapping exists between the WS_Bypass template in the metadata model, and a (user-defined) register in the CTL information model.

B. Behavioral Rules Verifier

The behavioral verification is the most complex step of the verification process. Behavioral and mixed rules can only be validated using a behavioral approach. Differently from the syntax and semantic verification, the behavioral approach is based on a functional simulation of the core test wrapper and of the core itself. Simulation is the only effective approach to verify compliancy of time-related rules, protocols, signal connections, and correct instructions implementation. The IEEE Standard 1500 Verification Framework models the behavioral rule verifier with the *Behavioral_Rule_Verifier* class in Fig. 3. The class implements the *do_Verification()* method that actually performs the behavioral verification. The behavioral rule verifier needs to simulate the core/wrapper it therefore has a *Simulate* association with the *IEEE_1500_Compliant_Core* class (see Fig. 3). The IEEE_1500_Compliant_Core class models the core under verification (it usually corresponds to an HDL description of the core).

A well-known approach to perform this type of verification is the so-called dynamic, coverage-driven, constrained-random simulation functional verification.

The term *dynamic* refers to the fact that the verification patterns/stimulus are generated and applied to the design over a number of clock cycles, and the corresponding results are collected and compared against a reference/golden model. An EDA simulator is used both to compute the values of all signals during the simulation and to compare the expected values with the calculated ones.

Simple dynamic verification has a main drawback: only a subset of all possible behaviors can be verified in a time-bound simulation run. Testing all possible behaviors under every possible combination of input stimuli is in most of the cases an unfeasible task since the test space is too large to be fully covered in a reasonable amount of time.

To overcome this problem, the number of verification patterns applied to the wrapper has to be statistically significant but not complete. To do this, verification input patterns are *generated randomly* under a set of *constraints*, which are expressed as mathematical expressions limiting the set of legal values on the input signals that drive the design. In this way, the simulator generates random values and constraints ensure that the generated scenarios are valid and plausible.

To further optimize this constrained-random generation, *coverage-driven verification* is used. Functional coverage metrics are automatically and in real-time recorded in order to ascertain whether (and how effectively) a particular test verified (or is verifying) a given feature; this information can then be fed back into the generation process in order to drive additional verification efforts more effectively towards the required goal. The coverage metrics are evaluated on coverage monitoring points defined by the user. The market offers a number of tools that are able to support this dynamic (or functional) verification methodology. The most used ones are Specman Elite (Cadence) [15] and Vera (Synopsys) [16]. Besides the different verification and pattern generation engines, all of them apply the verification patterns to the target design using a verification component placed around the core under analysis. The verification component, which is a behavioral-level module described using a proprietary verification language (*e* for Specman Elite, *OpenVera* for Vera), performs the constrained-random generation of the verification patterns, applies them, and is directly controlled by the verification engine monitoring the current coverage reached in the verification process.

To efficiently apply this verification approach to perform the behavioral verification of IEEE Standard 1500 rules it is necessary to do the following.

- Create a *rule verification component* for each rule (or subset of similar rules). The rule verification component is in charge of generating the verification patterns that, applied during the simulation, will allow checking that all the architectural and behavioral aspects of the rule are correctly implemented in the design.
- Identify in the design the *rule coverage points* for that rule. A coverage point is a signal/register in the wrapper that needs to be monitored in order to evaluate the coverage reached during the verification process on that particular rule.

The concept of coverage applied to the IEEE Standard 1500 rules is very important. Not only allows to compute the number

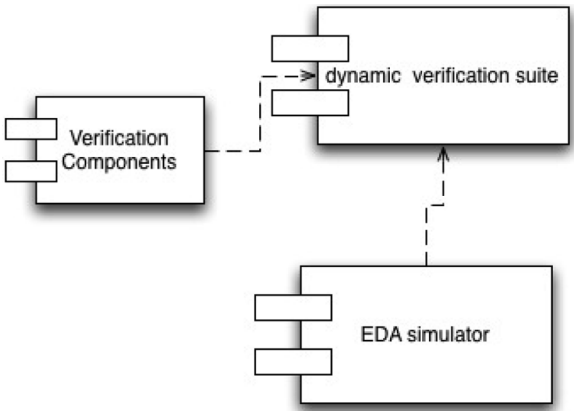

Fig. 9. IEEE 1500 behavioral verifier component diagram.

of verified rules over the set of available ones, but also allows to compute a compliancy level of each individual rule. As briefly explained before, if the combination of possible input patterns/internal states of the components involved in the rule is too large, the verification engine resorts to the constrained-random pattern generation. This will lead to the application of a subset of all possible patterns and therefore to a rule coverage or compliancy level possibly lower than 100%.

The most challenging issue in this phase is to write rule verification components and identify rule coverage points that are independent from the specific core or wrapper under analysis. Again, the name mapping stored in the metadata model described in Section V-A can be used to abstract the verification component from the specific core implementation.

Another very important issue to be considered in this phase is whether the core under verification is a black- or a white-box. The difference is in the amount of available information on the core's internal structure. In a black-box core the only available information is the input/output (I/O) interface. For IP protection the internal structure of a black-box core is unknown (except to the core designer, of course). A core integrator, who buys cores from different vendors, usually deals only with black-box cores. On the other hand, a core designer always has all the information regarding the core, and therefore uses the white-box approach.

From the IEEE Standard 1500 compliancy verification point of view, the difference between a black- and a white-box core directly impacts the degree of compliancy that can be verified, whereas in a white-box core, all the internal signals of the core can be controlled and/or observed and therefore all rules can be thoroughly verified, in a black-box core only the rules (or the portions of them) that do not require directly controlling or observing the core internal signals can be fully verified. Full IEEE Standard 1500 verification compliancy can only be achieved when dealing with white-box cores or with black-box cores implementing only the basic requirements of the standard.

This very important aspect of the behavioral verification is modeled in the IEEE Standard Verification framework by the *Black_Box_Core_Test_Wrapper* class and the *White_Box_Core_Test_Wrapper* class.

Fig. 9 finally summarizes the main components of the behavioral verifier using an UML components diagram.

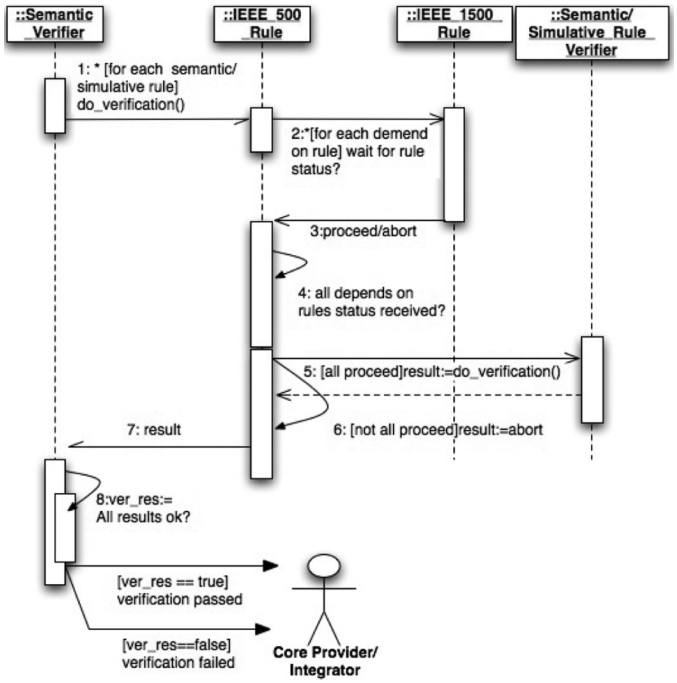

Fig. 10. Verification plan for semantic and behavioral rules.

C. Semantic and Behavioral Verification Plan

As already introduced in Section III, the definition of an efficient test plan is critical in reducing the overall verification costs. The sequence diagram of Fig. 4 introduced a first level of scheduling among the three different verification activities performed by the IEEE Standard Verification Framework. We can now enter into more details and model how these activities can be integrated together.

1) Syntax Verification Test Plan: The syntax verification is a one-step verification process. It just performs a single action that consists in parsing the CTL files provided with the IEEE Standard 1500 information model. No particular scheduling is needed for this verification.

2) Semantic and Behavioral Verification Test Plan: Both semantic and behavioral verification involve the verification of a set of IEEE Standard 1500 rules. The order in which the rules are verified is extremely important, first of all to guarantee the correctness of the result of the verification process and second to reduce the overall verification time by performing different actions in parallel.

The definition of the rule hierarchy described in Section V and modeled using the Depend_On association in the UML class diagram of Fig. 3 implicitly defines a verification plan for both semantic and behavioral rules. The plan is modeled by the UML sequence diagram of Fig. 10. The semantic/behavioral verifier sends a message to each rule object asking to perform the verification. The message is an asynchronous message (half arrow) that means that the verification of the full set of rules is performed in parallel.

Each rule, before performing its actual verification, first controls the status of the rules it is dependent on. It is possible to have the following three different situations:

- all rules are already verified: it is possible to proceed with the test;
- all rules are already verified but some of them failed: the whole verification process may be aborted or not, depending on the relationship severity of the involved rules;
- some rule is still working on its verification: the rule has to wait.

When all rules have completed the verification, the semantic/behavioral verifier collects the results and can inform the actor about the compliancy of the wrapper.

Obviously, the two types of verification (semantic/behavioral) will work on different sets of rules and the set of rules that will be verified will depend on the features actually implemented in the wrapper.

VI. Implementation

This section overviews a prototype implementation of the abstract UML model of the IEEE Standard 1500 Verification Framework presented in the previous sections. The full framework has been implemented using the Java [17] developing platform. The Java language is a full object oriented programming language and it is a perfect candidate to implement UML models.

Besides the general interface provided by the implementation, we will focus in this section on a possible implementation of the three types of verifiers (i.e., syntax, semantic, and behavioral), being these components the real core of the verification framework. As already introduced in Section V-B, the most critical component in the framework is the behavioral rule verifier. This component needs to perform a dynamic functional verification. Many commercial platforms are available to perform this type of verification; for our implementation, we used the Specman Elite [15] suite.

A. Syntax Verification

Several technologies allow the automatic implementations of parsers starting from the description of a formal grammar. We successfully implemented a syntax analyzer by using two open-source tools. JFLEX [18] is a lexical analyzer generator for Java, developed by V. Paxson. JFLEX is designed to work together with the parser generator CUP [19] by S. Hudson. The two tools allow the description of a formal language grammar (in our case the CTL grammar) and the automatic generation of a Java environment (collection of Java classes) able to perform the syntax analysis of a text file, according to the defined language. The definition of the set of lexical rules and of the grammar needed to implement the parser has been performed by analyzing the IEEE CTL definition [10] and by translating a set of informal definition contained in the standard into a formal definition suitable for the generation of the parsers.

Fig. 11 shows an example of how syntax definition of the IEEE Standard 1450.6 (CTL) are translated into formal definitions needed to generate the parser by using JFLEX and CUP. In the example, we have three types of informal rules: rules #1 and #2 describe with a simple, precise, and intuitive vocabulary how the functional rules will be and indicate directly the definition of a set of lexical definitions (i.e., "newline" and "tab"). Rule

INFORMAL LEXICAL RULES

```
1)Whitespace: space tab newline_character statement_teminators
2)Comments: Line comment(line comments are terminated by newline)
Block comment(block comment may span multiple lines)
3)Identifier:"to allow complex design names is to enclose the names
within double quotes, that may contain any Printable ASCII
character,including blanks,tabs...."
```

FORMAL LEXICAL RULES

```
1)WhiteSpace = {LineTerminator} | [ \t\f]
2)CommentContent = ( [^*] | \*+ [^/*] )*
Comment={TraditionalComment}|{EndOfLineComment}|
{DocumentationComment }
3) }...(([a-zA-Z]|" ")([a-zA-Z]|[0-9]|" ")* | \"( [a-zA-Z] | [0-9] | " " |
"!" | "@" |"#" | "$" | "%" | "^" | "&" | "*" | "(" | ")" | "-" | "+" | "="
| "|" | "`" | "{" | "}" | "[" | "]" | ":" | ";" | "¢" | "," | ">" |
"<" | "." | "?"| "/" | \\ | " " | [\t] | [\n] )* \" )(\.(([a-zA-Z]|" )([a-
zA-Z]|[0-9]|" ")*| \"( [a-zA-Z] | [0-9] | " " | "!" | "@" | "#" | "$" | "%"
| "^" | "&" | "*" |"(" | ")" | "-" | "+" | "=" | "|" | "⊥" | "{" |
"}" | "[" | "]" | ":" |";" | "¢" | "," | ">" | "<" | "." | "?" | "/" | \\
| " " | [¥t] | [¥n] )* ¥"))* {
set current line(); return symbol(sym.IDENTIFIER,yytext());
```

Fig. 11. Lexical rules example.

#3 is more complex: formally, it gives a global surface description without providing, for instance, what type of characters are allowed or not. To formally describe these types of rules, an extensive analysis of the IEEE Standard 1450.6 has been necessary.

B. Semantic Verification

The implementation of the semantic verification mainly consists in the implementation of the metadata structure defined in Section V-A. We implemented the metadata as a collection of Java classes as defined by the UML class diagram in Fig. 8. The task of populating the metadata with the actual CTL information is demanded to the CTL parser implemented in Section VI-A. Finally, each IEEE Standard 1500 semantic rule has been translated into a set of queries performed on the metadata. Actually, this type of implementation is not the more efficient one. Due to the number of rules to verify and to the complexity of a real core, the memory size and the complexity of the metadata structure in memory may become very high. For a commercial implementation, the use of a data base management system (DBMS) would be recommended.

C. Behavioral Verification

The implementation of the behavioral verification process relies on the use of the Specman Elite verification environment [15] interfaced with the Synopsys VCS simulator [20]. Specman Elite allows the definition of functional verification plans by using an object oriented language named *e*. *e* is a complete verification language that allows the definition of the following:

- built-in data generation from objects definition and constraints;
- notation of time like HDL simulators;
- built-in parallel execution;
- HDL interface; read from and write to HDL signals at any hierarchical level; call HDL tasks;
- predefined verification capabilities; automates handling checks without having to write complex procedural code;

- predefined flow execution.

In addition, it defines a so-called *e-Reuse Methodology* (eRM) aiming at designing reusable, consistent, extensible, plug-and-play verification environments, and *"e"* Verification Components (eVCs). In the implementation of our prototype, we followed this design methodology.

D. Application Cases

The functionalities of the IEEE Standard 1500 Verification Framework prototype have been tested on a simple IEEE Standard 1500 compliant core implementing a four bit counter with the following characteristics:

- CLOCK input used as counting clock;
- RESET input to reset the counting state;
- LOAD input to force a new start value for the counting;
- 4-bit input DIN that indicates the start value used when LOAD is high;
- 4-bit output COUNT that indicates the actual counting value.

This core has been wrapped with a IEEE Standard 1500 core test wrapper having the following characteristics:

- instruction register (WIR), 3 bit length;
- bypass register (WBY), 1 bit length;
- boundary register (WBR), 8 bit length;
- optional TransferDR wrapper serial control;
- four implemented instructions: WS_BYPASS, WS_PRELOAD, WS_INTEST, WS_EXTEST.

The verification process run on the wrapper/core was successful from the very beginning, when it allowed to discover design bugs we involuntarily introduced the wrapper design.

In order to carefully validate the verification capabilities of the prototype, we generated a set of different core test wrappers, systematically violating different rules of the standard. As an example, Fig. 12 shows the result of the behavioral verification in case of a non-compliant wrapper. In the example, rule number 10.2.1.c fails. The prototype highlights this violation and also provides the waveform obtained by the simulation to provide a better understanding of the reasons that led to the rule violation. On going work is focusing on creating a violation-programmable wrapper, where different violations can be enabled or disabled in order to verify the efficiency of the verification framework also in presence of multiple violations in the same wrapper.

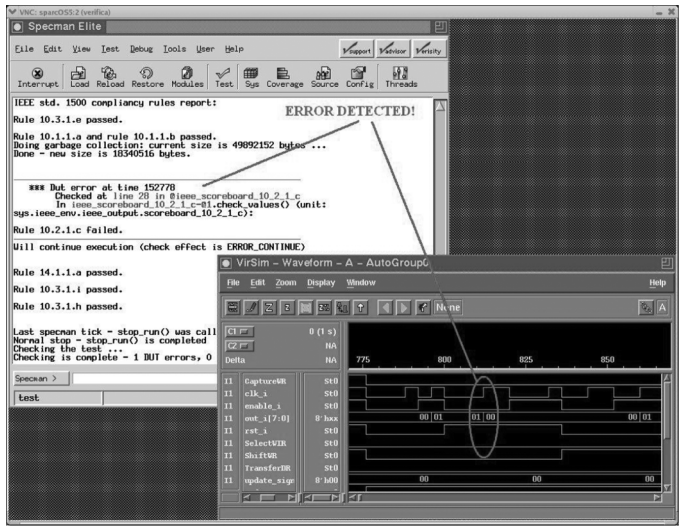

Fig. 12. Behavioral rule violation example.

VII. Conclusion

In this paper, we presented a systematic methodology and an associated formal model to build a verification framework for IEEE Standard 1500 compliant cores. The framework is able to check if the implementation of the wrapper provided with an IP core correctly follows the architectural and behavioral rules defined in the IEEE Standard 1500. The proposed framework targets different possible users, from the core designer to the core integrator, and therefore is able to guarantee various level of compliancy depending on the amount of information about the internal core structure available to the user. We also presented a proof-of-concept of the proposed model implementing a prototype of the verification framwork in Java and with the support of the Specman Elite verification toolkit. We believe that in the near future, with the introduction of IEEE Standard 1500 compliant wrappers in all IP cores on the market, a verification framework following the model presented in this paper will be able to increase productivity, reduce design time, and optimize the test plan of very complex SoCs.

Acknowledgment

The authors would like to thank D. Scollo, G. Politano, A. Mouth, and L. Melchionda for their help in the development of this manuscript.

References

[1] R. Gupta and Y. Zorian, "Introducing core-based system design," *IEEE Des. Test. Comput.*, vol. 14, no. 2, pp. 15–25, Oct. 1997.

[2] Y. Zorian, E. Marinissen, and S. Dey, "Testing embedded-core-based system chips," *IEEE Computer*, vol. 32, no. 6, pp. 52–60, Jun. 1999.

[3] *IEEE Standard Test Access Port and Boundary-Scan Architecture*, IEEE Std. 1149.1, 1990.

[4] *IEEE Standard Testability Method for Embedded Core-Based Integrated Circuits*, IEEE Std. 1500, 2005.

[5] L. Jin-Fu, H. Hsin-Jung, C. Jeng-Bin, S. Chih-Pin, W. Cheng-Wen, S.-I. C. C. Cheng, H. Chi-Yi, and L. Hsiao-Ping, "A hierarchical test methodology for systems on chip," *IEEE Micro*, vol. 22, no. 5, pp. 69–81, Sep. 2002.

[6] Y. Zorian, "Test requirements for embedded core-based systems and IEEE p1500," in *Proc. IEEE Int. Test Conf. (ITC)*, Nov. 1997, pp. 191–199.

[7] T. McLaurin and S. Ghosh, "Etm10 incorporates hardware segment of ieee p1500," *IEEE Des. Test. Comput.*, vol. 19, no. 3, pp. 6–11, May 2002.

[8] S. Picchiotino, M. Diaz-Nava, B. Forest, S. Engels, and R. Wilson, "Platform to validate soc designs and methodologies targeting nanometer cmos technologies," in *Proc. IP-SOC*, Dec. 2004, pp. 39–44.

[9] I. Diamantidis, T. Oikonomou, and S. Diamantidis, "Towards an IEEE verification infrastructure a comprehensive approach," in *Proc. ClubV*, Mar. 2005, p. 1500.

[10] *IEEE Core Test Language*, IEEE Std. 1450.6, 2005.

[11] Object Management Group, Needham, MA, "UML official website," 2008 [Online]. Available: http://www.uml.org/

[12] Object Management Group, Needham, MA, "Object management group official website," 2008 [Online]. Available: http://www.omg.org/

[13] N. Wirth, *Compiler Construction*. Reading, MA: Addison-Wesley, 1996.

[14] *Standard Test Interface Language (STIL) Standard Test Interface Language (STIL) Standard Test Interface Language (STIL) Standard Interface Test Language (STIL)*, IEEE Std. 1450.0, 1999.

[15] Cadence, San Jose, CA, "Specman elite home page," 2008 [Online]. Available: http://www.verisity.com/products/specman.html
[16] Synopsys, Mountain View, CA, "Vera Web Site," 2008 [Online]. Available: http://www.synopsys.com/products/vera/vera.html
[17] Sun Microsystems, "Java official website," 2008 [Online]. Available: http://www.java.com
[18] Gerwin Klein, "Jflex home page," 2008 [Online]. Available: http://jflex.de/index.html
[19] GVU Center, Georgia Institute of Technology, Atlanta, "Cup home page," 2008 [Online]. Available: http://www2.cs.tum.edu/projects/cup/
[20] Synopsys, Mountain View, CA, "Vcs home page," [Online]. Available: http://www.synopsys.com/products/simulation/simulation.html

Alfredo Benso (SM'07) currently holds a tenured Associate Professor position with the Department of Computer Engineering, Politecnico di Torino, Torino, Italy, where he teaches Microprocessor Systems and Advanced Programming Techniques. In his scientific career, mainly focused on hardware testing and dependability, he coauthored more than 60 publications between books, journals, and conference proceedings. He is also actively involved in the Computer Society, where he has been the leading volunteer for several projects such as the Technical Committees Archives (TECA) database, and Conference Information Management Application (CIMA).

Prof. Benso is a Computer Society Golden Core Member.

Stefano Di Carlo (M'03) received the M.S. degree in computer engineering and the Ph.D. degree in information technologies from Politecnico di Torino, Torino, Italy, in 1999 and 2004.

He is an Assistant Professor with the Department of Computer Engineering, Politecnico di Torino. His research interests mainly focus on DFT techniques, SoC testing, BIST, and memory testing. He coauthored more than 30 publications between journals and conference proceedings.

Prof. Di Carlo is a Golden Core Member of the IEEE Computer Society.

Paolo Prinetto received the M.S. degree in electronic engineering from Politecnico di Torino, Torino, Italy.

He is a Full Professor with the Department of Computer Engineering, Politecnico di Torino and a Joint Professor with the University of Illinois, Chicago. His research interests include testing, test generation, BIST, and dependability.

Prof. Prinetto is a Golden Core Member of the IEEE Computer Society and he has served on the IEEE Computer Society TTTC: Test Technology Technical Council as an elected chair.

Yervant Zorian (F'99) received the M.Sc. degree from the University of Southern California, Los Angeles, and the Ph.D. degree from McGill University, Montreal, QC, Canada.

He is the Vice President and Chief Scientist of Virage Logic Corporation, Fremont, CA, and an Adjunct Professor with the University of British Columbia, Vancouver, BC, Canada. He was previously a Distinguished Member of the technical staff with AT&T Bell Laboratories and Chief Technology Advisor of LogicVision. He served as the IEEE Computer Society Vice President for Conferences and Tutorials, Vice President for Technical Activities, Chair of the IEEE Test Technology Technical Council, and Editor-In-Chief of the IEEE DESIGN AND TEST OF COMPUTERS. He has authored over 300 papers, holds 16 U.S. patents.

Dr. Zorian was a recipient of numerous Best Paper Awards, a Bell Labs' R&D Achievement Award, the 2005 prestigious IEEE Industrial Pioneer Award, and the 2006 IEEE Hans Karlsson Award. He was selected by EE Times among the top 13 influencers on the semiconductor industry.